**Pintos Project 1: User Program (1)**

담당 교수 : 김영재 교수님

조 / 조원 : 박태윤

개발 기간 : 3주

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

pintos환경에서 argument를 user stack에 passing하고 이를 통해 system call을 수행한다. 해당 프로젝트에선 halt, exit, exec, wait, read, write와 추가적인 시스템 콜로 fibonacci와 max\_of\_four\_int를 수행하는 additional을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

pintos/src/userprog의 process.c에서 수행한다. load함수에서 argument를 parsing하여 user stack에 스택 포인터인 esp를 이용, 해당 argument의 내용과 저장된 주소를 스택에 저장한다.

1. User Memory Access

스택을 참조할 때 is\_user\_vaddr함수를 이용하여 kernel영역을 침범하지 않도록 pintos/src/userprog syscall.c에서 처리를 해준다. 유저 메모리의 스택이라면 syscall.c에서 read, write등 system call함수를 수행한다.

1. System Calls

hex\_dump함수를 이용하여 load함수에서 passing한 argument에 대한 정보가 스택의 어느 위치에 쌓여있는지를 확인하고 syscall.c에서 f->esp값을 적절하게 참조하여 system call함수의 인자로 넘겨준다. 수행 결과는 f의 eax레지스터에 저장을 해준다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

우선 userprog의 process.c안의 load함수에서 user stack에 argument를 passing한다. argument는 load함수의 인자인 file\_name로 들어온다. 예를 들어 ‘echo x’라고 argument가 들어오면 이를 띄어쓰기 기준으로 잘라 table이라는 배열에 table[0] = “echo”, table[1] = “x”와 같이 저장해준다. 이 과정에서 arg\_num이라는 변수를 이용하여 인자의 개수를 지정해준다.(arg\_num = 2)이 과정이 parsing이고 parsing이후에 user stack에 passing을 진행하는데 이 때 esp\_table이라는 배열을 이용하여 stack에 인자들이 저장되는 위치를 저장해준다. 처음에 \*esp를 minus시켜서 memcpy(\*esp, table[idx], tmp\_len)과 같이 스택에 인자들을 저장해준다.(tmp\_len은 해당 인자 길이 + 1이다)이 과정에서 esp\_table[] = \*esp를 통해 저장된 위치를 esp\_table에 넘겨준다. 이후 4칸으로 맞춰주는 alignment를 진행하고 그 다음에 인자가 저장된 주소를 가지고 있는 esp\_table의 내용을 스택에 넣어준다. 이 과정이 끝나면 인자 주소 저장이 완료된 주소의 시작점을 스택에 넣어주고 인자의 개수인 arg\_num을 스택에 넣어준다. user stack에 passing된 내용을 이용하여 이를 참조해 kernel stack에 쌓아주고 이를 통해 system call을 진행한다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
  + Invalid memory access를 어떻게 막을 것인지 설명

user mode에서 kernel의 메모리 영역을 참조하려고 하는 것이 invalid memory access이다. invalid memory access는 esp값과 커널 메모리의 시작점인 PHYS\_BASE간의 비교를 통해서 제어할 수 있는데 이 과정은 userprog의 syscall.c에서 각 system call함수를 호출하기 이전 is\_user\_vaddr함수를 이용해 이 함수의 값이 false라면(커널 영역을 참조하고 있음)system call을 진행하지 않고 exit(-1)를 통해 종료시켜주었다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

user mode에서 kernel의 영역을 삭제 및 수정하면 안되는데, 이를 막으면 kernel안의 유용한 api들을 사용하지 못하게 된다. 따라서 이를 사용하게 하는 것이 system call이다.

* + 1. SYS\_HALT

현재 pintos를 종료시키는 시키는 시스템 콜이다. halt함수를 이용하여 구현하였으며 이 안에서 shutdown\_power\_off()함수를 이용하여 동작시켰다.

* + 1. SYS\_EXIT

현재 쓰레드를 종료시키는 시스템 콜이다. exit함수를 이용하여 구현하였으며 이 안에서 thread\_exit()함수를 이용하여 쓰레드를 종료시켰다.

* + 1. SYS\_EXEC

실행을 위한 새로운 쓰레드를 만들어주는 시스템 콜이다. exec함수를 통해 구현하였으며 이 함수는 process\_execute함수를 호출하고 이 안에서 쓰레드를 생성한다.

* + 1. SYS\_WAIT

부모 쓰레드가 자식 쓰레드보다 먼저 종료되지 않도록 해주는 시스템 콜이다. wait함수를 이용하여 구현하였으며 이 함수는 process\_wait함수를 호출하여 wait동작을 실행한다.

* + 1. SYS\_READ

열려있는 파일을 읽는 시스템 콜이다. read함수를 통해 구현하였으며 인자로 받은 buf에 파일의 데이터를 저장해준다.

* + 1. SYS\_WRITE

파일에 데이터를 써주는 시스템 콜이다. write함수를 통해 구현하였으며 현재는 STDOUT에 해당하는 file descriptor에 대해서만 화면에 내용을 출력할 수 있도록 구현하였다.

* + 1. SYS\_FIBO

추가적으로 구현한 Fibonacci 시스템 콜이다. fibonacci함수를 통해 동작을 하며 인자로 받은 숫자에 대한 피보나치 수를 반환하는 함수이다.

* + 1. SYS\_MAX

추가적으로 구현한 max\_of\_four\_int 시스템 콜이다. max\_of\_four\_int함수를 통해 동작을 하며 4수 중 가장 큰 수를 반환하는 함수이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

1주차 : argument parsing 및 passing 구현

2주차 : 시스템 콜 구현

3주차 : make check를 이용하여 결과를 체크한 뒤

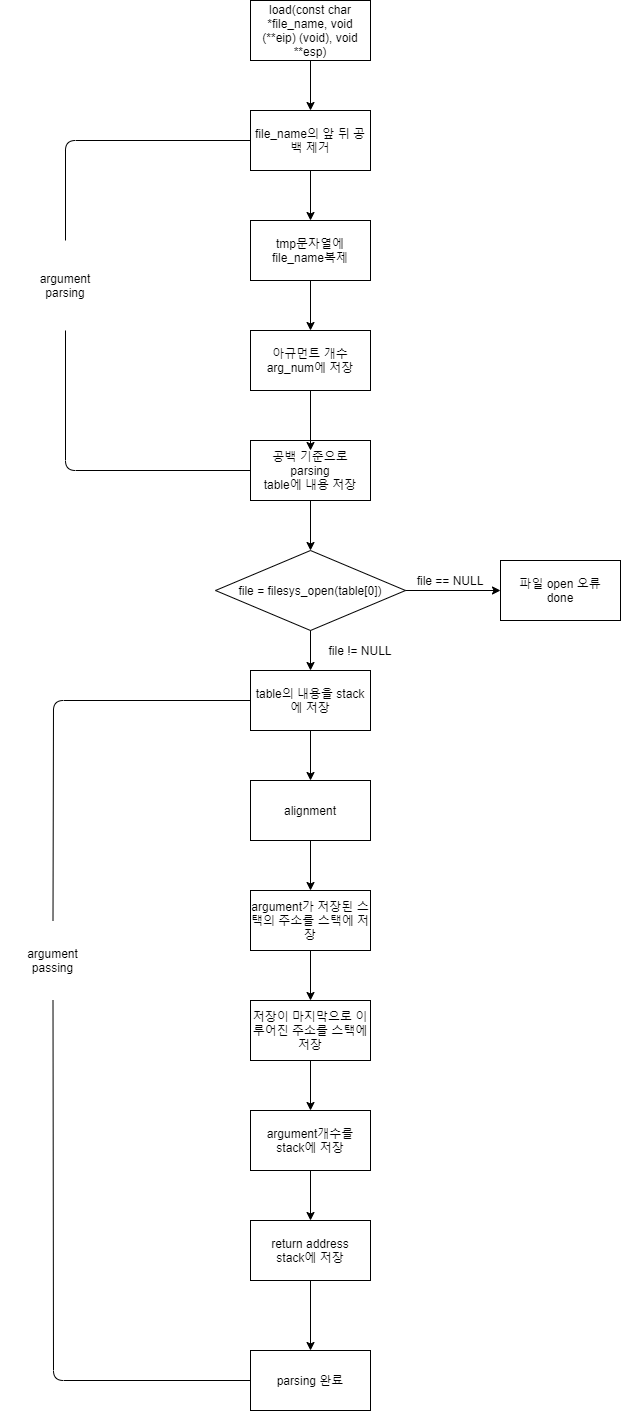
* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

추가적인 함수로는 우선 userprog디렉토리의 syscall.h에 작성되어있는 함수에 따라 system call함수(wait, read 등)을 syscall.c에 구현하였다. parsing과정에서는 문자열의 공백들을 모두 ‘ ‘(space)로 바꿔주는 changeBlank와 인자로 받은 문자열의 앞 뒤 공백을 지워주는 eraseBlank함수를 구현하였다. 또한 wait동작을 위해 생성된 쓰레드의 정보를 가지는 thread\_list구조체를 threads디렉토리의 thread.h에 구현하였는데, 이 구조체는 생성된 쓰레드의 쓰레드 아이디를 나타내는 tid, 부모 쓰레드의 아이디를 나타내는 parent\_tid, 동작 상태(1이면 동작 중, 0이면 동작 종료, -100이면 exit)를 나타내는 flag, thread\_exit시에 exit\_status를 나타내는 exitStatus를 가진다. 이를 thread\_list wait\_list[200]이라는 배열을 생성하여 관리를 해주었으며, thread\_create에서 쓰레드를 생성할 때 마다 배열에 요소를 추가하였다. 이 과정에서 create\_idx라는 전역 변수를 thread.h에 추가하였는데, 추가될 때 마다 1씩 증가하며 exit를 할 때마다 1씩 줄어들도록 구현하였다. 추가적으로, 해당 쓰레드 아이디가 wait\_list에서 몇 번째 인자로 표현이 되는지를 나타내는 search\_wait\_list, 자식 쓰레드 유무를 나타내는 there\_is\_no\_child\_thread, wait\_list의 정보를 출력해주는 print\_wait\_list, 남아있는 쓰레드의 유무를 나타내는 there\_is\_no\_remain, status로 -100을 가져 아직 사라지지 않은 쓰레드의 유무를 나타내는 there\_is\_no\_zombie함수를 추가하였다. 또한 추가적인 시스템 콜을 수행할 수 있는 additional.c를 examples에 추가하였으며 이에 대한 동작을 하는 fibonacci와 max\_of\_four\_int함수를 userprog디렉토리의 syscall.c에 구현하였다.

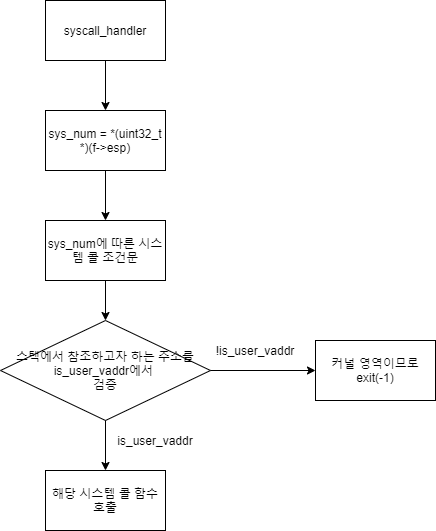
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

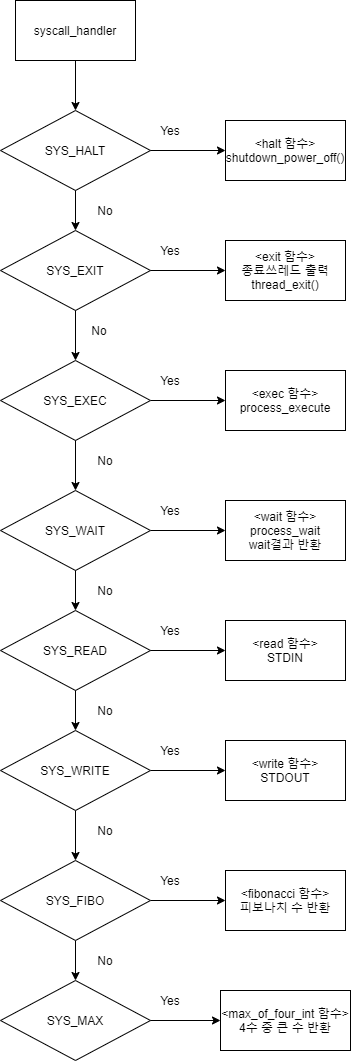
1. Argument Passing



1. User Memory Access



1. System Calls



* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

passing 이전에 공백을 기준으로 argument로 잘라주는 parsing과정이 필요하다.

eraseBlank(file\_name);

strlcpy(tmp, file\_name, strlen(file\_name)+1);

for(int i = 0; i < strlen(tmp); i++){

if(tmp[i] != ' ' && isWord == false){

arg\_num++;

isWord = true;

}

if(tmp[i] == ' ' && isWord == true){

isWord = false;

}

}

arg\_num--;

//printf("qqqqqqqqqqqqqqqqq %d\n",arg\_num);

//table에 첫번째 argument 복사

ptr = strtok\_r(tmp, " ", &next\_ptr);

strlcpy(table[0], ptr, strlen(ptr)+1);

eraseBlank(table[0]);

//table에 두번째 이후 arg parsing

for(int j = 1; j < arg\_num; j++){

ptr = strtok\_r(NULL, " ", &next\_ptr);

strlcpy(table[j], ptr, strlen(ptr)+1);

eraseBlank(table[j]);

table[j][strlen(table[j])] = '\0';

//printf("%s\n",table[j]);

}

table[0][strlen(table[0])] = '\0';

strlcpy(file\_name, table[0], strlen(table[0])+1);

eraseBlank로 인자로 받은 argument인 file\_name의 앞 뒤 공백을 지워주고 이를 parsing하기 위해 tmp라는 문자열에 내용을 복제해준다. 이후 argument의 개수를 세주는데, 공백 개수에 따라 argument의 개수를 나타내는 arg\_num을 하나씩 증가시켜준다. 이 때, 공백이 연속으로 올 수 있기 때문에 isWord라는 flag를 만들어 처리를 해주었다. 이후 문자열 포인터 ptr을 이용해 strtok\_r함수를 이용하여 공백을 기준으로 tmp문자열을 잘라준다. 이 내용을 char table[25][50]이라는 문자열에 하나씩 추가를 해주는데, 공백이 연속으로 오는 경우를 고려하여 eraseBlank로 자를 때 마다 앞 뒤 공백을 지워주면서 처리를 해준다. 문자열의 마지막 글자는 항상 \0이 올 수 있도록 따로 처리를 해주었다.

char \*\*esp\_table;

esp\_table = (char \*\*)malloc(sizeof(char \*) \* arg\_num);

int total\_len = 0;

int idx = arg\_num;

while(1){

idx--;

if(idx < 0)

break;

int tmp\_len = strlen(table[idx])+1;

total\_len += tmp\_len;

\*esp -= tmp\_len;

memcpy(\*esp, table[idx], tmp\_len);

//strlcpy(\*esp, table[idx], tmp\_len+1);

esp\_table[idx] = \*esp;

}

//hex\_dump(\*esp, \*esp, 100, 1);

if((PHYS\_BASE - \*esp) % 4 != 0){

\*esp -= sizeof(uint32\_t \*\*) - ((PHYS\_BASE - \*esp) % 4);

}

\*esp -= sizeof(uint32\_t \*\*);

\*\*(uint32\_t \*\*)esp = 0;

for(i = arg\_num-1; 0 <= i; i--){

\*esp -= sizeof(uint32\_t \*\*);

\*(uint32\_t \*\*)\*esp = esp\_table[i];

}

\*esp -= sizeof(uint32\_t \*\*);

\*(uint32\_t \*\*)\*esp = \*esp + 4;

\*esp -= sizeof(uint32\_t \*\*);

\*(uint32\_t \*\*)\*esp = arg\_num;

\*esp -= sizeof(uint32\_t \*\*);

\*(uint32\_t \*\*)\*esp = 0;

user stack에 argument내용을 passing해주는 코드이다. char \*\*esp\_table은 table의 argument가 스택에 저장된 주소를 담는 변수이고, total\_len 아규먼트의 총 길이를 나타낸다. while(1)문을 돌면서 처음에 아규먼트 개수를 나타내는 idx가 0미만이 될 때까지 아규먼트 내용을 스택에 올리는 작업을 한다. memcpy(\*esp, table[idx], tmp\_len)와 같이 스택에 내용을 올려주었으며, 예를 들어 echo x인 경우 x가 먼저 올라가고 이후 echo가 올라가야 하기 때문에 idx는 arg\_num에서부터 0까지 거꾸로 진행이 된다. 이 과정 속에서 스택에 올라가는 주소를 알아야 이후 스택에 아규먼트가 저장된 주소를 스택에 올려줄 수 있기 때문에 esp\_table[idx] = \*esp로 처리를 해주었다. \*esp는 현재 저장한 문자열의 길이만큼 빼주면서 스택에 올렸다. 이후 alignment를 처리하여 4칸으로 맞춰주었고 esp\_table을 이용하여 스택에 문자열이 담긴 주소를 스택에 쌓아주었다. 마지막으로 지금까지 스택에 내용을 쌓아준 시작 지점을 스택에 올려주었으며 아규먼트의 개수인 arg\_num을 쌓아주었다.

1. User Memory Access

void

syscall\_init (void)

{

intr\_register\_int (0x30, 3, INTR\_ON, syscall\_handler, "syscall");

}

static void

syscall\_handler (struct intr\_frame \*f UNUSED)

{

//printf ("system call!\n");

struct thread \*t = thread\_current();

// hex\_dump(f->esp, f->esp, 100, 1);

//hex\_dump(f->esp-100, f->esp-100, 100, 1);

int sys\_num = \*(uint32\_t \*)(f->esp);

if(sys\_num < 0){

printf("not valid system call number\n");

exit(-1);

}

else{

if(sys\_num == (int)SYS\_HALT){

//printf("SYS\_HALT\n");

//shutdown\_power\_off();

halt();

}

else if(sys\_num == (int)SYS\_EXIT){

if(!is\_user\_vaddr(f->esp + 4))

exit(-1);

//wait\_list[s\_idx].exitStatus = cur->exit\_status;

exit(\*(uint32\_t \*)(f->esp + 4));

}

else if(sys\_num == (int)SYS\_EXEC){

//printf("SYS\_EXEC\n");

if(!is\_user\_vaddr(f->esp + 4))

exit(-1);

f->eax = exec((const char \*)\*(uint32\_t \*)(f->esp + 4));

}

else if(sys\_num == (int)SYS\_WAIT){

//printf("SYS\_WAIT\n");

if(!is\_user\_vaddr(f->esp + 4))

exit(-1);

//hex\_dump(f->esp, f->esp, 300, 1);

f->eax = wait((pid\_t)\*(uint32\_t \*)(f->esp + 4));

}

else if(sys\_num == (int)SYS\_CREATE){

//printf("SYS\_CREATE\n");

}

else if(sys\_num == (int)SYS\_REMOVE){

//printf("SYS\_REMOVE\n");

}

else if(sys\_num == (int)SYS\_OPEN){

//printf("SYS\_OPEN\n");

}

else if(sys\_num == (int)SYS\_FILESIZE){

//printf("SYS\_FILESIZE\n");

}

else if(sys\_num == (int)SYS\_READ){

//printf("SYS\_READ\n");

//f->eax = i

if(!is\_user\_vaddr(f->esp + 20) || !is\_user\_vaddr(f->esp + 24) || !is\_user\_vaddr(f->esp+28))

exit(-1);

f->eax = read((int)\*(uint32\_t \*)(f->esp+20), (void \*)\*(uint32\_t \*)(f->esp + 24), (unsigned)\*((uint32\_t \*)(f->esp + 28)));

}

else if(sys\_num == (int)SYS\_WRITE){

//f-> eax = write(1, (void \*)\*(uint32\_t \*)(f->esp + 24), (unsigned)\*((uint32\_t \*)(f->esp + 28)));

if(!is\_user\_vaddr(f->esp + 24) || !is\_user\_vaddr(f->esp+28))

exit(-1);

//print("%d\n",(int)\*((uint32\_t\*)(f->esp + 28)));

f->eax = write((int)\*(uint32\_t \*)(f->esp + 20), (char \*)\*(uint32\_t \*)(f->esp + 24), (unsigned)\*((uint32\_t \*)(f->esp + 28)));

//putbuf((void \*)\*(uint32\_t \*)(f->esp + 24), (unsigned)\*((uint32\_t \*)(f->esp + 28)));

}

else if(sys\_num == (int)SYS\_SEEK){

//printf("SYS\_SEEK\n");

}

else if(sys\_num == (int)SYS\_TELL){

//printf("SYS\_TELL\n");

}

else if(sys\_num == (int)SYS\_CLOSE){

//printf("SYS\_CLOSE\n");

}

else if(sys\_num == (int)SYS\_FIBO){

//hex\_dump(f->esp, f->esp, 300, 1);

if(!is\_user\_vaddr(f->esp + 32))

exit(-1);

f->eax = fibonacci((int)\*(uint32\_t \*)(f->esp + 32));

}

else if(sys\_num == (int)SYS\_MAX){

//hex\_dump(f->esp, f->esp, 300, 1);

if(!is\_user\_vaddr(f->esp + 32) || !is\_user\_vaddr(f->esp + 36) || !is\_user\_vaddr(f->esp + 44))

exit(-1);

f->eax = max\_of\_four\_int((int)\*(uint32\_t \*)(f->esp + 32), (int)\*(uint32\_t \*)(f->esp + 36), (int)\*(uint32\_t \*)(f->esp + 40), (int)\*(uint32\_t \*)(f->esp + 44));

}

}

userprog디렉토리의 syscall.c안에 있는 syscall\_handler이다. 시스템 콜을 수행할 때 스택을 참조하는데, 각 시스템 콜 마다 참조하고자 하는 위치가 user영역인지를 검사를 해줌으로써 user memory access문제를 처리하였다. 만약 해당 위치에서 is\_user\_vaddr함수가 0을 리턴하면 이는 kernel영역을 참조하고 있다는 얘기이므로 exit(-1)로 프로그램을 종료시켰다.

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

int fibonacci(int n){

if(n == 0)

return 0;

else if(n == 1)

return 1;

else if(n == 2)

return 1;

else

return fibonacci(n-1) + fibonacci(n-2);

}

int max\_of\_four\_int(int a, int b, int c, int d){

int arr[4];

int max = 0;

arr[0] = a; arr[1] = b; arr[2] = c; arr[3] = d;

for(int i = 0; i < 4; i++){

if(max < arr[i])

max = arr[i];

}

return max;

}

int write(int fd, const void \*buffer, size\_t n){

//printf("%d\n",n);

if(fd == 1){

putbuf(buffer, n);

//printf("%s",(char \*)buffer);

return true;

}

else

return false;

}

void halt(void){

shutdown\_power\_off();

}

int read(int fd, void \*buffer, size\_t n){

int res;

if(fd == 0){

res = input\_getc();

return res;

}

else

return -1;

//return res;

}

pid\_t exec(const char\* command){

return process\_execute(command);

}

int wait (pid\_t process\_id){

int res = process\_wait(process\_id);

//if(res > -100)

return res;

}

void exit(int status){

printf("%s: exit(%d)\n", thread\_name(), status);

thread\_current() -> exit\_status = status;

int s\_idx = search\_wait\_list(thread\_current()->tid);

//thread\_current()->parent->child\_exit\_status = status;

wait\_list[s\_idx].exitStatus = status;

thread\_exit();

}

userprog디렉토리의 syscall.c안에 있는 시스템 콜 함수들이다. fibonacci함수는 추가적인 시스템 콜인 SYS\_FIBO에서 쓰이는 함수로 인자로 받은 n번째의 피보나치 수를 반환해주는 함수이다. max\_of\_four\_int역시 추가적인 시스템 콜인 SYS\_MAX에서 쓰이는 함수로 인자로 받은 a,b,c,d중 가장 큰 값을 반환해주는 함수이다. write는 현재 파일 디스크립터가 1인 STDOUT일 때만 화면에 내용을 putbuf를 이용해 출력할 수 있도록 구현하였다. halt는 SYS\_HALT시스템 콜에서 pintos를 종료시켜줄 수 있도록 shutdown\_power\_off함수를 이용하여 구현해주었고, read함수는 SYS\_READ시스템 콜에서 동작하는 함수로 현재는 STDIN에서만 동작하도록 구현하였다. exec함수는 process\_execute를 호출하는데, process\_execute안에서는

strlcpy(tmp, file\_name, strlen(file\_name)+1);

ptr = strtok\_r(tmp, " ", &next\_ptr);

strlcpy(command, ptr, strlen(ptr)+1);

if(filesys\_open(command) == NULL)

return -1;

와 같이 filesys\_open이 인자로 동작 아규먼트(echo x에서는 echo)를 받을 수 있게 parsing하는 과정을 따로 추가하였다.

wait함수는 process\_wait를 호출하는데,

int

process\_wait (tid\_t child\_tid)

{

struct list\_elem\* e;

struct list\_elem\* k;

struct thread\* t = NULL;

struct thread\* m = NULL;

int exit\_status;

int ttid;

// while(1){

//print\_wait\_list();

t = thread\_current();

int cur\_idx = search\_wait\_list(t->tid);

//if(there\_is\_no\_child\_thread(t->tid) == false){

while(1){

thread\_yield();

if(there\_is\_no\_child\_thread(t->tid)==true && there\_is\_no\_zombie() == true)

break;

}

// }

if(there\_is\_no\_child\_thread(t->tid) == true && create\_idx > 2){

//printf("%d\n",create\_idx);

for(int j = 1; j < create\_idx; j++){

if(wait\_list[j].flag == 0 && wait\_list[j].parent\_tid == t->tid){

ttid = wait\_list[j].tid;

exit\_status = wait\_list[j].exitStatus;

//what??

wait\_list[j].flag = -100;

wait\_list\_remove(j);

//break;

}

else if(wait\_list[j].flag == -1){

//printf("sssssssssssssssssssssssssssssssssssss\n");

return -1;

}

else if(wait\_list[j].flag == -100){

//printf("ttttttttttttttttttttttttttttttt\n");

continue;

}

}

//printf("%d %d\n",ttid,exit\_status);

//print\_wait\_list();

return exit\_status;

}

else if(there\_is\_no\_remain() == true){

//printf("haha\n");

//exit\_status = wait\_list[0].exitStatus;

return -1;

}

/\*

else{

while(1){

thread\_yield();

if(there\_is\_no\_child\_thread(t->tid)==true)

break;

}

}\*/

// }

return -1;

}

process\_wait함수는 busy waiting방식으로 구현하였다. 이는 thread\_yield()를 이용하였는데, 현재 쓰레드가 프로세스 제어권을 놓고 ready\_list의 뒤쪽으로 들어가게 하도록 하는 이 함수를 자식 쓰레드가 존재하지 않을 때까지 돌려서 waiting을 할 수 있도록 구현하였다. 쓰레드가 죽을 때 exit\_status를 따로 저장을 하여 이 함수에서 죽은 쓰레드의 exit\_status를 받아 return할 수 있도록 구현하였다. wait\_list에서 flag가 0이라는 소리는 동작 중인 경우에 flag가 1로 세팅이 되었다가 thread\_exit에서 걸리면 flag가 0으로 바뀌기 때문에 이는 곧 동작이 종료되었다는 것을 의미하여 이 때 해당 wait\_list[j].exitStatus를 exit\_status로 받아 마지막에 리턴을 해주었다. 이렇게 리턴을 해 준 wait\_list안의 쓰레드는 flag를 -100으로 세팅을 해주었다.

exit함수는 SYS\_EXIT시스템 콜이 걸릴 때 호출이 되는 시스템 콜 함수로 어떤 쓰레드가 exit가 되었는지를 printf로 화면에 출력한 뒤 인자로 받은 status를 해당 쓰레드의 exit\_status로, wait\_list에 해당 쓰레드 넘버에 해당하는 곳에 넣어주었다.

void

process\_exit (void)

{

struct thread \*cur = thread\_current ();

uint32\_t \*pd;

struct list\_elem\* e;

struct thread\* t = NULL;

// printf("exit : %d", cur->tid);

/\* Destroy the current process's page directory and switch back

to the kernel-only page directory. \*/

int s\_idx = search\_wait\_list(cur->tid);

//printf("num : %d\n",s\_idx);

pd = cur->pagedir;

if (pd != NULL && wait\_list[s\_idx].flag == -100)

{

/\* Correct ordering here is crucial. We must set

cur->pagedir to NULL before switching page directories,

so that a timer interrupt can't switch back to the

process page directory. We must activate the base page

directory before destroying the process's page

directory, or our active page directory will be one

that's been freed (and cleared). \*/

wait\_list[s\_idx].exitStatus = cur->exit\_status;

//printf("exit : %d\n", (int)thread\_tid());

if(there\_is\_no\_child\_thread(cur->tid) == false)

wait\_list[s\_idx].flag = -1;

else

wait\_list[s\_idx].flag = 0;

cur->pagedir = NULL;

pagedir\_activate (NULL);

pagedir\_destroy (pd);

}

}

쓰레드가 종료될 때 process\_exit함수에서 걸리는데 이 때 자식 쓰레드가 있는 경우에도 exit가 걸렸다면 kernel에 의해 종료가 된 것이므로 해당 wait\_list의 flag내용을 -1로 세팅을 해주었고 정상 종료라면 0으로 세팅을 해주었다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**

examples디렉토리에 additional.c파일을 만들었다.

#include <stdio.h>

#include <stdlib.h>

#include <syscall.h>

int main (int argc, char \*\*argv)

{

int i;

int arr[4];

for (i = 0; i < argc; i++){

arr[i] = atoi(argv[i]);

}

printf("%d %d\n", fibonacci(arr[0]), max\_of\_four\_int(arr[0], arr[1], arr[2], arr[3]));

return EXIT\_SUCCESS;

}

인자로 받은 argument를 모두 정수형으로 바꿔 arr배열에 저장한 뒤 fibonacci와 max\_of\_four\_int의 리턴 값을 출력할 수 있도록 구현하였다.

또한 lib/user의 syscall.h에

int fibonacci(int n);

int max\_of\_four\_int(int a, int b, int c, int d);

와 같은 추가적인 시스템 콜 동작을 위한 함수를 선언하였다.

max\_of\_four\_int는 인자가 4개인 시스템 콜 함수이기 때문에 syscall4를 syscall.c에 담음과 같이 선언을 해주었다.

#define syscall4(NUMBER, ARG0, ARG1, ARG2, ARG3) \

({ \

int retval; \

asm volatile \

("pushl %[arg3]; pushl %[arg3]; pushl %[arg1]; pushl %[arg0]; " \

"pushl %[number]; int $0x30; addl $20, %%esp" \

: "=a" (retval) \

: [number] "i" (NUMBER), \

[arg0] "r" (ARG0), \

[arg1] "r" (ARG1), \

[arg2] "r" (ARG2), \

[arg3] "r" (ARG3) \

: "memory"); \

retval; \

})

int fibonacci(int n){

return syscall1 (SYS\_FIBO, n);

}

int max\_of\_four\_int(int a, int b, int c, int d){

return syscall4 (SYS\_MAX, a,b,c,d);

}

또한 다음과 같이 인자 개수에 맞추어 syscall을 넣어주었다.

#ifndef \_\_LIB\_SYSCALL\_NR\_H

#define \_\_LIB\_SYSCALL\_NR\_H

/\* System call numbers. \*/

enum

{

/\* Projects 2 and later. \*/

SYS\_HALT, /\* Halt the operating system. \*/

SYS\_EXIT, /\* Terminate this process. \*/

SYS\_EXEC, /\* Start another process. \*/

SYS\_WAIT, /\* Wait for a child process to die. \*/

SYS\_CREATE, /\* Create a file. \*/

SYS\_REMOVE, /\* Delete a file. \*/

SYS\_OPEN, /\* Open a file. \*/

SYS\_FILESIZE, /\* Obtain a file's size. \*/

SYS\_READ, /\* Read from a file. \*/

SYS\_WRITE, /\* Write to a file. \*/

SYS\_SEEK, /\* Change position in a file. \*/

SYS\_TELL, /\* Report current position in a file. \*/

SYS\_CLOSE,

SYS\_FIBO,

SYS\_MAX, /\* Close a file. \*/

/\* Project 3 and optionally project 4. \*/

SYS\_MMAP, /\* Map a file into memory. \*/

SYS\_MUNMAP, /\* Remove a memory mapping. \*/

/\* Project 4 only. \*/

SYS\_CHDIR, /\* Change the current directory. \*/

SYS\_MKDIR, /\* Create a directory. \*/

SYS\_READDIR, /\* Reads a directory entry. \*/

SYS\_ISDIR, /\* Tests if a fd represents a directory. \*/

SYS\_INUMBER /\* Returns the inode number for a fd. \*/

};

#endif /\* lib/syscall-nr.h \*/

또한 다음과 같이 syscall-nr.h에 새로운 시스템 콜 넘버인 SYS\_FIBO와 SYS\_MAX를 추가하였다 이는 각각 fibonacci와 max\_of\_four\_int에 해당된다.

userprog의 syscall.c에서는 시스템 콜 함수를 다음과 같이 호출한 후 eax레지스터에 결과를 넣어주었다.

else if(sys\_num == (int)SYS\_FIBO){

//hex\_dump(f->esp, f->esp, 300, 1);

if(!is\_user\_vaddr(f->esp + 32))

exit(-1);

f->eax = fibonacci((int)\*(uint32\_t \*)(f->esp + 32));

}

else if(sys\_num == (int)SYS\_MAX){

//hex\_dump(f->esp, f->esp, 300, 1);

if(!is\_user\_vaddr(f->esp + 32) || !is\_user\_vaddr(f->esp + 36) || !is\_user\_vaddr(f->esp + 44))

exit(-1);

f->eax = max\_of\_four\_int((int)\*(uint32\_t \*)(f->esp + 32), (int)\*(uint32\_t \*)(f->esp + 36), (int)\*(uint32\_t \*)(f->esp + 40), (int)\*(uint32\_t \*)(f->esp + 44));

}

구체적인 구현 내용은 다음과 같다.

int fibonacci(int n){

if(n == 0)

return 0;

else if(n == 1)

return 1;

else if(n == 2)

return 1;

else

return fibonacci(n-1) + fibonacci(n-2);

}

int max\_of\_four\_int(int a, int b, int c, int d){

int arr[4];

int max = 0;

arr[0] = a; arr[1] = b; arr[2] = c; arr[3] = d;

for(int i = 0; i < 4; i++){

if(max < arr[i])

max = arr[i];

}

return max;

}

피보나치는 재귀 함수를 이용하여 구현을 해주었으며 max\_of\_four\_int는 아규먼트를 모두 arr배열에 넣어 대소관계를 반복문을 통해 비교해가며 가장 큰 값을 리턴하였다.

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

**텍스트이(가) 표시된 사진

자동 생성된 설명**